

目录

Introduction	1.1
Docs	1.2
Ch 00 Forword And Introduction	1.2.1
Ch 01 Getting Started	1.2.2

Rust 编程语言

原作者: *Steve Klabnik 与 Carol Nichols, 及 Rust 社区*

此版本的教材, 假定安装了 Rust 1.61.0 或更新版本。请参阅 [第 1 章的“安装”小节](#) 进行安装, 或对已安装的 Rust 进行升级。

```
$rustc --version
rustc 1.61.0 (Arch Linux rust 1:1.61.0-1)
```



在 [Gitbook](#) 上可阅读此教程: [Rust 编程语言](#)。

Copyright @ rust.xfoss.com 2022 all right reserved, powered by Gitbook该文件修订时间: 2022-06-14 12:37:21

前言和简介

虽然这样说有些含糊其辞，但基本上可说 Rust 编程语言，是一种 *赋能* (empowerment)：不管你当前在用哪种语言编写代码，Rust 都可以赋予你更大能力，在编程之路上走得更远，在你先前的各种领域，更具信心地编写程序。

就拿涉及到底层的内存管理、数据表示及并发等“系统层面”的工作来讲。传统上，这类编程都被认为是深奥的、只由极少数花费多年时间掌握了避开其间臭名昭著陷阱的程序员来完成。而即使这些人，在写这类代码时，仍然是小心翼翼，以免他们写出的代码留下漏洞利用、崩溃或不当。

通过消除原有各种陷阱，以及提供到友好、全新工具集，Rust 破除了编写这类苛刻程序中的诸多障碍。那么那些需要“深入”到底层控制的程序员们，现在就可以运用 Rust，在不必承担一直以来的崩溃或者安全漏洞的情况下，同时还无须去深入细致地掌握那变化无常工具链，就可以达成他们的目标了。更了不起的是，在设计这门语言时，就贯彻了引导使用他的程序员编写出可靠又高效的代码，体现在运行速度及内存使用上。

正在进行底层代码编写的程序员们，可运用 Rust 来提升他们的雄心壮志。比如说，在 Rust 中引入并行机制，是相对低风险的操作：编译器会为你捕获到那些经典错误。同时还可以在确信不会带来程序崩溃或漏洞利用之下，大胆进行更多的优化。

然而 Rust 并非局限于底层系统的编写。对于构造命令行应用、web 服务器及其他类别的代码来说，Rust 的表现力和人机工程设计，也可以让这些编写起来相当愉悦 -- 在本书后面，就会发现这样的示例。运用 Rust 可实现将一个领域中学到的技能，迁移到另一领域；通过编写 web 应用，就可以掌握 Rust，然后将这同样的技能应用到树梅派 app 的编写上。

本书充分接纳到 Rust 给其使用者赋能的潜力。这本书友好而恰当，试图帮助你不仅提升有关 Rust 的知识，还在一般方面提升你编程的水平和信心。那么就请继续阅读下去，欢迎来到 Rust 社区！

-- Nicholas Matsakis 与 Aaron Turon

简介

欢迎来到 *Rust 编程语言*，一本 Rust 的介绍性书籍。Rust 编程语言帮助更快地编写出更可靠软件。在程序语言设计中，上层人机交互与底层控制，通常是不可调和的；Rust 挑战了这对矛盾。经由强力的技术能力与了不起的开发者体验，Rust 带来了对底层细节（譬如内存的使用）控制的同时，免去了传统上底层控制带来的一大堆麻烦。

Rust 适用于哪些人群

对于相当多的人群，Rust 因为各种原因，都是理想选择。下面就来看看那些最重要群体中的一些人的情况。

开发者团队

对于有着不同水平系统编程知识的开发者团队的协同来讲，Rust 正被证明是一种生产力工具。底层代码倾向于出现各种细微错误，这样的细微错误，对于其他编程语言，则只能经由广泛测试，和经验老道的开发者细致代码评审才能捕获到。而在 Rust 中，编译器通过拒绝编译这些难以捉摸的错误，包括并发错误，而扮演着看门人角色。通过与编译器一道工作，团队就可以将他们的时间，集中用在程序逻辑，而不是找寻错误上。

Rust 还带给了系统编程世界，一些现代开发者工具：

- `cargo`，Rust 所包含的依赖管理器与构建工具，让整个 Rust 生态中添加依赖、编译与管理依赖，变得愉快并具一致性（`cargo`，the included dependency manager and build tool, makes adding, compiling, and managing dependencies painless and consistent across the Rust ecosystem）；
- `Rustfmt` 确保了不同开发者之间有着一致的编码风格；
- Rust 语言服务器驱动了用于代码补全与行内错误消息的集成开发环境。

通过使用这些开发者工具，及其他一些 Rust 生态中的工具，开发者就可以在编写系统级代码时，颇具生产力了。

学生

Rust 是为学生及那些对掌握系统概念感兴趣的人所准备的。运用 Rust，许多人都掌握了像是操作系统开发这样的知识点。Rust 社区非常欢迎并乐于回答学生们提出的问题。通过像是本书这样的努力，Rust 团队本身是要让更多人，尤其是那些刚开始编程的人们，可获取到系统概念。

商业公司

已有上千家规模或大或小的商业公司，在生产中，为着不同任务使用着 Rust。这些任务包括了命令行工具、web 服务、运维工具、嵌入式装置、音视频分析与转码、加密货币、生物信息学、搜索引擎、物联网应用、机器学习，甚至 Firefox web 浏览器的主要部分等等。

开放源代码开发者

Rust 是为着那些想要构建 Rust 编程语言本身、Rust 社区、Rust 开发者工具和库而准备的。我们希望你为 Rust 语言做出贡献。

看重运行速度与稳定性的人们

Rust 是为那些渴求某门语言所提供速度与稳定性的人们准备的。这里说的运行速度，指的是使用 Rust 可创建出程序的运行速度，以及 Rust 所能达到的编写这些程序速度。Rust 编译器的检查，确保了功能补充与重构的稳定性。这稳定性是与那些不具备这些检查语言中的脆弱老旧代码相比得出的，开发者通常害怕去修改那些脆弱老旧代码。通过争取实现零代价的抽象，就有了那些在手动编写时，就立即编译到底层代码的上层特性，Rust 致力于实现在构造安全代码的同时，还取得了快速的代码编写与程序运行。

Rust 语言也希望带给众多其他用户以支持；这里提到的只是一些最大的相关群体。总体来讲，Rust 最伟大的抱负，是要消除程序员们在过去数十年来，业已被迫接受的在安全性与生产力、开发和运行速度及人机交互上的妥协。请给 Rust 一个机会，然后看看 Rust 的选择是否适合于你。

本书读者群体

本书假定你曾编写过其他某种编程语言的代码，至于何种编程语言并不重要。本书作者已尽力让其中的教学材料适合到有着宽泛编程背景的读者。这里不会花大量时间来解释编程为何物，以及该怎么来看待编程。若对编程一窍不通，那么最好找一本编程入门的书先看看。

如何使用本书

大体上，本书假定是要以从前往后的顺序进行阅读。后续章节是建立在较早章节的概念之上，而较早的章节不会深入某个话题的细节；在后续章节通常会回顾到那个话题。

本书有两种章节：概念性章节与项目性章节。在概念章节，将对 Rust 某个方面的有所了解。而在项目性章节，就会构建出一些在一起的小程序，这些小程序运用了概念性章节中学到的东西。第 2、12 和 20 章，就是项目性章节；而剩下的，全都是概念性章节。

第 1 章讲了怎样安装 Rust、怎样编写出“Hello, world!”程序，还有怎样使用 Rust 的包管理器及构建工具 `cargo`。第 2 章是 Rust 语言的一个实操介绍。这里涵盖了上层的一些概念，而在后续章节则会提供到进一步的细节。若要立即动手编写代码，那么第 2 章就可以开始了。一开始你或许想要跳过第 3 章，这一章涵盖了与那些其他编程语言类似的一些 Rust 特性，而要直接前往到第 4 章，去了解 Rust 的所有权系统。不过若要细致了解 Rust，就应详细掌握 Rust 的每个细节设计之后，在移步到下一章节，或许也会跳过第 2 章直接到第 3 章，然后在想要将学到的细节知识应用到项目时，再回到第 2 章。

第 5 章讨论了结构体和方法，同时第 6 章涵盖了枚举、`match` 表达式，和 `if let` 控制流结构。在构造 Rust 中的定制类型时，就会用到这些结构体和枚举。

第 7 章将了解到 Rust 的模组系统，以及代码组织的隐私规则，还有 Rust 的公共应用编程接口（Application Programming Interface, API）。第 8 章讨论了一些常用的、由标准库提供的集合数据结构，诸如矢量、字符串即哈希映射。第 9 章探索了 Rust 的错误处理思想与技巧。

第 10 章涉及了范型、特质（traits）与生命周期，他们赋予了定义出应用多种类型代码的能力。第 11 章全都是关于测试的内容，即便有着 Rust 的安全性保证，对于确保程序逻辑正确，测试仍是不可缺少的。第 12 章将构建一个我们自己的、用于在文件中搜索文本的 `grep` 命令行工具功能子集的版本。到这里，就会用到先前章节中所讨论的众多概念了。

第 13 章对闭包（closures）和迭代进行了探索：闭包和迭代属于 Rust 来自函数式编程语言的特性。第 14 章，将更深入地对 `cargo` 加以检视，并就如何与他人分享库的最佳实践进行探讨。第 15 章讨论标准库提供一些灵巧指针，还有实现不同功能的 Rust 特质（traits）。

第 16 章，将遍数并发编程的各种不同模型，并探讨 Rust 如何对大胆进行多线程编程的帮助。第 17 章将或许你所熟知面向对象编程的那些原则，与 Rust 下编程的习惯加以对比。

第 18 章是模式与模式匹配的一个参考，这在 Rust 程序中，属于是概念表达的强大方式。第 19 章包含了一个诸多感兴趣的话题大杂烩，包括不安全的 Rust、宏，以及更多有关生命周期、特质（traits）、类型、函数与闭包等等的细节。

第 20 章，将完成一个其中实现了底层线程化的 web 服务器！

最后，还有一些包含了这门语言的有用信息的附录，这些附录则更多的像是参考的形式。附录 A 涵盖了 Rust 的关键字，附录 B 涵盖了 Rust 的运算符和符号，附录 C 涵盖了由标准库所提供的那些派生特质（derivable traits），附录 D 涵盖了一些有用的开发工具，还有附录 E 对 Rust 版本进行了解释。

阅读本书并无定法：你要跳着去读，也是可以的！在遇到疑惑时，或许就不得不跳回去看看了。你只要怎么有效就行了。

掌握 Rust 过程中的一个重要部分，就是要学会怎样去读那些编译器给出的错误消息：这些错误消息将引导你得到运作的代码。由此，本书将提供到许多不编译的示例，以及在各种情况下编译器将给出的错误消息。请知悉在进入到某个随机示例并加以运行时，示例代码可能会不编译！请确保要阅读这些示例周围的文字，来了解正尝试运行的示例，是不是有错误。本书中的虚拟人物 `Ferris` 也会帮助你识别代码是否会工作的：

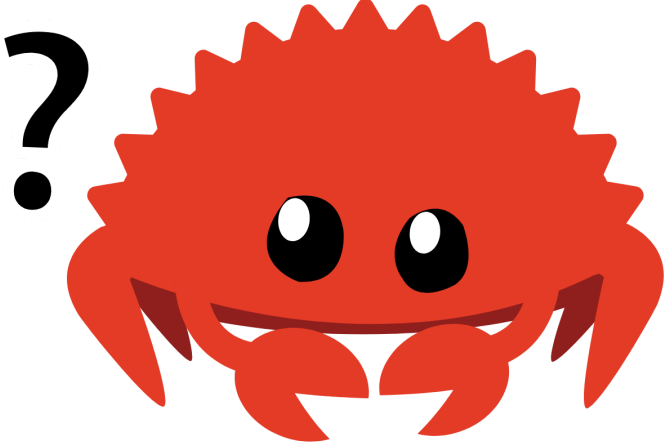
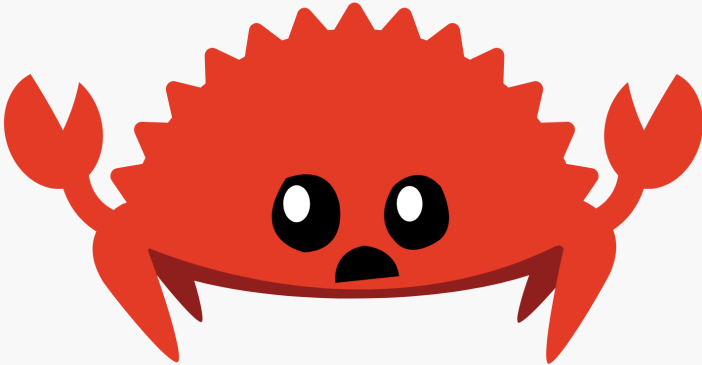
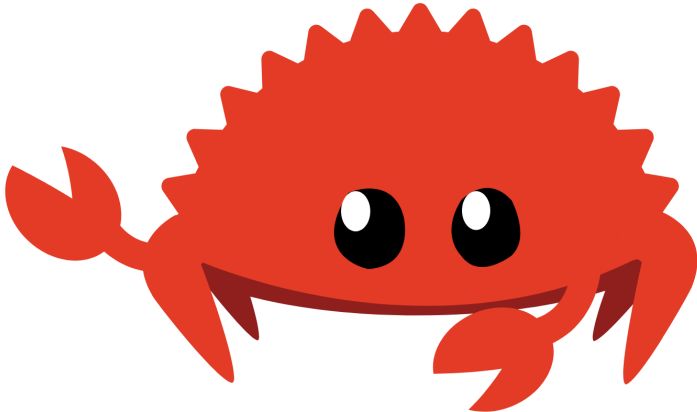
Ferris	意义
	此代码不会编译!
	此代码不会运行!
	此代码不会产生出期望的行为。

表 1 - Ferris 表情含义

多数情况下，这里都会给出不会编译代码的正确版本。

本书的源码

本书所产生的源码，可在 [Github: gnu4cn/rust-lang](https://github.com/gnu4cn/rust-lang) 下载到。

Copyright @ rust.xfoss.com 2022 all right reserved, powered by Gitbook该文件修
订时间: 2022-06-14 12:29:37

入门

现在就开始 Rust 之旅！有很多要掌握的东西，不过千里之行，始于足下。本章将讨论：

- 在 Linux、macOS 及 Windows 上安装 Rust;
- 编写一个打印出 `Hello, world!` 的程序来；
- Rust 的包管理器和构建系统 Cargo 的使用。

安装

第一步即是安装 Rust。这里将通过 `rustup` 这个用于管理 Rust 版本及相关工具的命令行动工具，来下载 Rust。要下载 Rust，就需要互联网连接。

注意：若由于某些原因而不愿使用 `rustup`，那么请参考 [其他 Rust 安装方式页面](#) 了解更多选项。

接下来就是要按照最新的稳定版 Rust 编译器。Rust 的稳定性保证了本书中所有示例都将在较新的 Rust 版本下可持续编译。由于 Rust 经常会改进错误消息和告警，因此在不同版本之间，输出可能会略有不同。也就是说，任何使用以下步骤所安装的较新、稳定版 Rust，都将如本书内容中所期望的那样工作。

关于**命令行注释** 在本章及全书中，都会给出一些在终端中用到的命令。他们是一些应在以 `$` 开始的终端中输入的行。至于这个 `$` 字符，是无需输入的；这个字符表示每条命令的开头。那些不以 `$` 开头的行，通常给出的是上一命令的输出。此外，那些特定于 `PowerShell` 的示例中，将使用 `>` 而不是 `$`。

在 Linux 与 macOS 上安装 `rustup`

若使用的是 Linux 或 macOS，那么请打开一个终端，然后输入下面的命令：

```
$ curl --proto 'https' --tlsv1.2 https://sh.rustup.rs -sSf | sh
```

此命令会下载一个脚本并开始 `rustup` 工具的安装，而 `rustup` 将安装最新的稳定版 Rust。可能会提示输入 `sudo` 密码。在安装成功后，就会出现下面这行！

```
Rust is installed now. Great!
```

这里还将需要一个连接器（linker），这是个 Rust 要用来将其编译好的输出，组合起来形成一个文件的程序。似乎你的电脑上以及有了一个这样的连接器了。若收到连接器错误信息，那么就应安装一个 C 语言编译器，C 编译器通常会包含着连接器的。由于一些常用 Rust 包对 C 代码有依赖且需要 C 编译器，因此 C 编译器也是有用的。

在 macOS 上，可通过运行下面的命令，获取到一个 C 编译器：

```
$ xcode-select --install
```

Linux 用户一般都会安装 GCC 或 Clang，至于具体哪种 C 编译器，则是依据他们所用 Linux 分发版本的文档可以确定。比如若使用的是 Ubuntu，那么就可以安装 `build-essential` 软件包。

在 Windows 上安装 `rustup`

在 Windows 上，请前往 <https://www.rust-lang.org/tools/install> 页面，并按照安装 Rust 的指令进行安装。在安装过程的某个时刻，将收到为何需要 Visual Studio 2013 或更新版本的 C++ 构建工具的说明。而最简单的获取到构建工具的方法，则是安装 [Visual Studio 2019 构建工具](#)。在询问将要安装何种工作负载（workloads）时，请确保 `C++ build tools` 被选中，还要确保包含 Windows 10 SDK 及英语语言包。

本书接下来用到的命令，在 `cmd.exe` 与 `PowerShell` 中都可工作。若其中有特定区别，本书将会解释要用哪个。

更新与卸载

在通过 `rustup` 安装了 Rust 后，更新到最新版本就容易了。在 `shell` 中运行下面的更新脚本：

```
$ rustup update
```

而要卸载 Rust 和 `rustup`，只需在 `shell` 中运行下面的卸载脚本：

```
$ rustup self uninstall
```

故障排除

要检查当前是否安装了 Rust，请开启一个 `shell` 并敲入这行命令：

```
$ rustc --version
```

就会看到版本编号、合并哈希（`commit hash`），以及已发布的该最新稳定版本合并日期，以下面这种格式：

```
rustc x.y.z (abcabcadc yyyy-mm-dd)
```

若看到这个信息，那么就已成功安装了 Rust！若看不到这个信息，且是在 Windows 上，那么就请在 `%PATH%` 系统变量中检查一下 Rust 在不在里面。若那一点问题都没有而 Rust 仍就不工作，那么可在数个地方需求帮助。其中最便利的就是 [Rust 官方 Discord](#) 上的 `#beginners` 频道了。在那里可与其他 Rust 公民（一种无厘头的自我称呼）聊天，他们可以帮助你。其他不错的资源包括 [用户论坛](#) 和 [Stack Overflow](#)。

本地文档

Rust 的安装，也包含了一份本地文档，因此可离线阅读到这本地文档。运行 `rustup doc` 即可在浏览器中打开这本地文档。

在任何时候遇到标准库所提供的类型或函数，而又确定他做些什么或该怎样使用这类类型或函数时，就可以使用 API 文档来搞明白他是怎么回事！

Hello, World!

既然已经安装好了 Rust, 那么就编写第一个 Rust 程序吧。在掌握一门新语言时，传统就是要编写一个小的、打印出文字 `Hello, World!` 到屏幕上的程序，因此这里也会干这同样的事情！

注意：本书假定读者对命令行有着基本的熟悉。Rust 对代码在何处编辑和使用何种工具编辑没有特别要求，因此若优先选择某种集成开发环境，而非命令行，那么使用喜好的 IDE 即可。许多 IDE 都有某种程度的 Rust 支持；请查看 IDE 文档了解有关细节信息。近来，Rust 团队已着手启动良好的 IDE 支持，且此方面已取得极大进展！

创建一个项目目录

这里是以构造一个保存 Rust 代码的目录开始的。对于 Rust 来说，代码位居何处并不重要，不过对于本书中的练习与项目，是建议在主目录下构造一个 `projects` 目录，并把全部项目放在那里的。

请打开一个终端，并输入下面的这些命令来构造一个 `projects` 的目录，和一个在 `projects` 下用于 "Hello, World!" 项目的目录。

对于 Linux、macOS 和 Windows 上的 PowerShell，请输入：

```
$ mkdir ~/rust-lang/projects
$ cd ~/rust-lang/projects
$ mkdir hello_world
$ cd hello_world
```

而对于 Windows 的 CMD，请输入：

```
> mkdir "%USERPROFILE%\rust-lang\projects"
> cd /d "%USERPROFILE%\rust-lang\projects"
> mkdir hello_world
> cd hello_world
```

编写及运行 Rust 程序

接下来，就要构造一个源代码文件，并命名为 `main.rs`。Rust 文件总是以 `.rs` 扩展名结束。若要在文件名中是一个单词，那么请使用下划线来将这些单词隔开。比如，请使用 `hello_world.rs` 而不是 `helloworld.rs`。

现在就要打开这个刚创建出的 `main.rs` 文件，并敲入清单 1-1 中的代码。

文件名： `main.rs`

```
fn main() {
    println!("Hello, World!");
}
```

清单 1-1: 打印 `Hello, World!` 的程序

保存这个文件并回到终端窗口。在 Linux 或 macOS 上，请输入下面的命令来编译和运行这个文件：

```
$ rustc main.rs
$ ./main
Hello, World!
```

在 Windows 上，就要输入命令 `.\main.exe` 而不是 `./main`：

```
> rustc main.rs
> .\main.exe
Hello, World!
```

而不论所在操作系统为何，字符串 `Hello, World!` 都应打印到终端。而若没有看到这个输出，那么请回到安装小节的“故障排除”部分获取帮助。

如确实打印出了 `Hello, World!`，那么恭喜你！你已正式编写除了一个 Rust 程序了。那就让你成为了一名 Rust 程序员了 -- 欢迎！

Rust 程序解析

来仔细回顾一下刚才在“Hello World！”程序中发生了什么。这是谜团中第一部分：

```
fn main() {

}
```

这些行定义了 Rust 中的一个函数。这个 `main` 函数比较特殊：在每个可执行的 Rust 程序中，他总是第一个开始运行的代码。这第一行声明了一个名为 `main` 的、没有参数且不返回任何值的参数。若函数有参数，那么参数就应位处圆括号 `()` 内部。

还有就是，请注意函数体是包裹在花括号 `{}` 中的。Rust 要求将全部函数体都用花括号包裹起来。将开头的花括号与函数声明放在同一行，并在二者之间加上一个空格，是良好的代码风格。

若想要在多个 Rust 项目之间保持一种标准的编码风格，那么就可以使用一个名为 `rustfmt` 的自动格式化工具，来以一种特定样式对代码进行格式化。与 `rustc` 一样，Rust 团队已将此工具包含在标准的 Rust 发布中，因此你的电脑上就应该已经有了这个格式化工具了！请查看在线文档了解更多详情。

在这个 `main` 函数里头，是下面的代码：

```
println!("Hello, World!");
```

这行代码完成了此小程序的全部工作：他将文字打印到屏幕。这里有四个需要注意的重要细节。

首先，Rust 编码风格是缩进四个空格，而非一个制表符；

其次，`println!` 调用了 Rust 的宏（a Rust macro）。若他调用的是个函数，那么就应输入 `println`（是不带 `!` 的）。在后续的第 19 章，将详细讨论 Rust 的宏。而现在，则只需知道 `!` 的使用表示是在调用某个宏而不是普通函数，同时宏不会总是遵循与函数同样的规则；

第三，就是看到的 `Hello, World!` 这个字符串了。这里时将此字符串作为参数，传递给 `println!` 的，且这个字符串是被打印到屏幕上的；

最后，这行语句是以分号（`;`）结束的，这表示该表达式结束，同时下一表达式已准备好开始。Rust 代码的多数行，都是以分号结束的。

编译和运行是分开的步骤

这里刚刚运行了一个新近创建出的程序，那么来检视一下该过程的每个步骤。

在运行某个 Rust 程序之前，必须要通过敲入 `rustc` 命令并将源代码文件名字，作为 `rustc` 的参数加以传入，这样来使用 Rust 编译器对其进行编译，像下面这样：

```
$ rustc main.rs
```

如果你有 C 或 C++ 的背景知识，那么就会注意到这与 `gcc` 或 `clang` 类似。在成功编译后，Rust 就会输出一个二进制可执行文件。

在 Linux、macOS 和 Windows 上的 PowerShell 之上，就可以通过在 `shell` 中敲入 `ls` 看到这个可执行文件。在 Linux 与 macOS 上，将看到下面这两个文件。而在 Windows 上的 PowerShell 中，则会看到与使用 CMD 一样的以下三个文件。

```
$ ls
main    main.rs
```

在 Windows 的 CMD 中，就应输入下面的东西：

```
> dir /B %* 这里的 /B 选项表示只显示文件名 =%
main.exe
main.pdb
main.rs
```

这显示了带有 `.rs` 扩展名的源代码文件、那个可执行文件（Windows 上的 `main.exe`，对于其他平台则是 `main`），以及，在使用 Windows 时，一个包含了调试信息的、带有 `.pdb` 扩展名的文件。从此处，就像下面这样来运行这里的 `main` 或 `main.exe`：

```
$ ./main # 或在 Windows 上的 .\main.exe
```

若这里的 `main.rs` 就是那个“Hello, World!”程序，那么这行命令就会将 `Hello, world!` 打印到你的终端了。

若你对某门动态语言，诸如 Ruby、Python 或者 JavaScript 更为熟悉，那么可能就不习惯于将编译和运行某个程序作为分开的步骤。Rust 是门 *提前编译语言*（an *ahead-of-time compiled language*），这意味着可对程序进行编译，而将可执行文件交给他人，他们可在未安装 Rust 的情况下运行编译好的可执行文件。而若将某个 `.rb`、`.py`，或者 `.js` 文件交给某人时，他们就需要安装好相应的 Ruby、Python 或 JavaScript 实现。不过在这些语言中，仅需一个命令来编译和运行他们的程序。在编程语言设计中，每件事都有所取舍。

对于简单的程序来说，用 `rustc` 编译就足够了，但随着项目的成长，就希望对所有选项进行管理，并令到代码分享更为简便。接下来，就要介绍 Cargo 工具了，这工具将帮助我们编写出实用的 Rust 程序。

你好，Cargo！

Cargo 是 Rust 的构建系统和包管理器。由于 Cargo 处理了很多任务，诸如构建代码、下载代码所依赖的库，以及这些库的构建等等，因此绝大多数 Rust 公民都使用这个工具，来管理他们的 Rust 项目。（我们把这些库叫做代码需要依赖（we call the libraries that your code needs *dependencies*）。）

对于最简单的那些 Rust 程序，比如才写的那个，是没有任何依赖的。因此若使用 Cargo 来构建这个 `Hello, World!` 项目，那么就只会用到 Cargo 处理代码构建的部分。而随着更为复杂 Rust 程序的编写，就会添加依赖，而在开始一个用到 Cargo 的项目时，完成依赖添加就会容易得多。

由于广大 Rust 项目都用到了 Cargo，本书其余部分就假定了也使用 Cargo。若使用了在 [安装](#) 小节中提到的官方安装器进行的 Rust 安装，那么 Cargo 就已与 Rust 一起安装好了。而若是以其他方式安装的 Rust，那么就要通过在终端中敲入下面的命令，来检查 Cargo 是否已安装妥当：

```
$ cargo --version
```

若能看到版本号，那么就拥有了这个工具！而若看到错误，诸如 `command not found`，就请查看你的安装方式的文档，找到怎样单独安装 Cargo 的方法。

使用 Cargo 创建项目

下面来使用 Cargo 创建一个新项目，并看看与原先的“Hello, World!”项目有何不同。现在导航至 `projects` 目录（或确定下来的保存代码的其他地方）。然后不论在那个操作系统之上，运行下面的命令：

```
$ cargo new hello_cargo
$ cd hello_cargo
```

这第一个命令创建出了一个新的名为 `hello_cargo` 目录。这里就已将项目命名为了 `hello_cargo`，然后 Cargo 将其文件创建在了同名的目录里面。

进入到 `hello_cargo` 目录并列出那些文件。就会看到 Cargo 已经为我们生成了两个文件和一个目录：一个 `Cargo.toml` 文件与一个里头有着 `main.rs` 文件的 `src` 目录。

`cargo new` 还初始化了一个新的、带有 `.gitignore` 文件的 Git 代码仓库。若是在一个已有的 Git 代码仓库运行的 `cargo new`，那么就不会生成那些 Git 文件；通过运用 `cargo new --vcs=git` 可重写此行为。

注意：Git 是种常用的版本控制系统。可通过上面的 `--vcs` 命令行参数，让 `cargo new` 使用其他版本控制系统或不使用版本控制系统。请运行 `cargo new --help` 命令来查看所有可用选项。

文件名： `Cargo.toml`

```
[package]
name = "hello_cargo"
version = "0.1.0"
edition = '2021'

[dependencies]
```

清单 1-2：由 `cargo new` 所生成的 `Cargo.toml` 的内容

该文件是 TOML（*Tom's Obvious, Minimal Language*）格式的，这是 Cargo 的配置格式。

该文件的第一行，`[package]`，是个小节标题，表示接下来的语句是在对一个包进行配置。随着往这个文件添加越来越多的信息，就会添加其他小节。

接下来的三行，对 Cargo 用于编译程序所需的信息进行了配置：项目名称、版本号及要使用的 Rust 版本。在 附录 E 中会讲到这个 `edition` 关键字。

`Cargo.toml` 的最后一行，`[dependencies]`，是要列出项目全部依赖小节开始的地方。在 Rust 中，代码包被称为包裹（*crates*）。此项目无需任何其他包裹，在第 2 章中的头一个项目，就会用到依赖包裹，因此在那时就会用到这个依赖小节。

现在打开 `src/main.rs` 然后看看：

文件名： `src/main.rs`

```
fn main() {
    println! ("Hello, World!");
}
```

Cargo 以及为我们生成了一个“Hello, World!”的程序，这个自动生成的程序就跟之前在清单 1-1 中的一样！到现在，先前的项目与这个 Cargo 生成的项目的不同之处，就是 Cargo 是将代码放在那个 `src` 目录中的，同时在顶层目录还有一个 `Cargo.toml` 配置文件。

Cargo 希望那些源代码文件，存留在 `src` 目录里头。而顶层的项目目录，只用于 `README` 文件、许可证信息、配置文件及其他与代码无关的东西。使用 Cargo 有助于对项目的组织。一切都有了个地方，且一切都在各自的地方（*there's a place for everything, and everything is in its place*）。

若没有使用 Cargo 来开始项目，就如同先前在“Hello, World!”项目中所做那样，那么仍旧可使用 Cargo 将其转换为一个项目。将项目代码移入到 `src` 目录并创建一个适当的 `Cargo.toml` 文件来：

```
$ cd hello_world
$ mkdir src
$ mv main.rs src/
$ cargo init
```

构建和运行一个 Cargo 项目

现在来看看在使用 Cargo 来构建和运行那个“Hello, World!”程序有什么不同之处！在 `hello_cargo` 目录，通过敲入下面的命令，来构建该项目：

```
$ cargo build
   Compiling hello_cargo v0.1.0 (/home/peng/rust-lang/projects/hello_cargo)
   Finished dev [unoptimized + debuginfo] target(s) in 0.45s
```

此命令创建出在 `target/debug/hello_cargo`（或 Windows 上的 `target\debug\hello_cargo.exe`）中，而非当前目录下的一个可执行文件。可使用下面这个命令运行那个可执行程序：

```
$ ./target/debug/hello_cargo # 或者在 Windows 上的 .\target\debug\hello_cargo.exe
Hello, world!
```

Copyright @ rust.xfoss.com 2022 all right reserved, powered by Gitbook该文件修订时间： 2022-06-14 12:29:37